

Algoritmos y Estructuras de Datos 2

Final 2022-08-02

Ejercicio 1

Explicar cómo afecta en el diseño que un TAD sea inconsistente, sobreespecificado y subespecificado.

Inconsistente

Un TAD es inconsistente cuando la axiomatización de alguna operación produce distintos resultados para la misma entrada. Recordemos que las axiomatizaciones no se “evalúan” en orden. Es decir, valen todos los axiomas al mismo tiempo para todas las posibles combinaciones de argumentos de entrada (no existe una evaluación secuencial top-down de los axiomas).

Por ejemplo, esta axiomatización produce una inconsistencia ya que la secuencia de entrada $3 \bullet S$ es evaluada en ambos axiomas, pero uno devuelve 21 y el otro 3.

```
operacion(3 • S) ≡ 21
operacion(a • S) ≡ a
```

Es por esto que conviene axiomatizar las operaciones de forma tal que los conjuntos de entradas para cada axioma sean disjuntos (si se axiomatiza sobre los generadores y son minimales, ésto sucede automáticamente). Si un TAD es inconsistente, entonces es imposible diseñar sus algoritmos ya que no sabemos cuál es el resultado correcto para la operación que generó la inconsistencia.

Sobreespecificado

Un TAD está sobreespecificado si tenemos varias formas de obtener el resultado de una operación para una determinada entrada. Es decir, tenemos axiomas “de más”. Esto es legal siempre y cuando el TAD sea consistente (llegamos al mismo resultado utilizando cualquier combinación de axiomas).

Por ejemplo, esta axiomatización está sobreespecificada pero es consistente. Con solo el segundo axioma sería suficiente.

```
operacion(3 • S) ≡ 3
operacion(a • S) ≡ a
```

El problema principal de sobreespecificar es que el TAD puede resultar confuso. Siempre conviene mantener los axiomas lo más minimal posible, y que haya una única forma inequívoca de aplicar los axiomas para obtener el resultado de alguna operación.

También sobreespecificamos cuando en nuestro TAD utilizamos algún otro tipo de dato que tiene ciertos comportamientos no relevantes para nuestro problema. Por ejemplo, si estamos modelando un carrito de supermercado que contiene productos, éstos podrían ser una secuencia o un conjunto. ¿Cuál conviene usar? Si usamos una secuencia estamos dándole un orden a los productos, pero si no importa el orden conviene usar un conjunto para permitir en el etapa de diseño más opciones de estructuras.

Subespecificado

Existen varias formas de subespecificar. La más común sucede cuando restringimos el dominio de alguna operación. Por definición de un TAD, todas las operaciones son totales (están definidas en todo su dominio). No obstante, es común que ciertas operaciones simplemente no tengan sentido para un subconjunto del dominio, ya sea porque es imposible determinar el resultado (por ejemplo dividir por 0) o porque no son casos relevantes en el contexto de uso (por ejemplo una operación que recibe como argumento un monto de dinero podría solo tener sentido si el monto es ≥ 0).

Cuando aplicamos una restricción sobre el dominio de una operación, lo que estamos haciendo es recortar el dominio a los valores relevantes para nuestro problema, y solo considerar esos valores en los axiomas, sin decir qué pasa con los valores restringidos. Esto simplifica la axiomatización. Durante la etapa de diseño, se debe decidir qué hacer al recibir una entrada restringida. Se podría simplemente no hacer nada si la operación se puede realizar de todas formas, o se fuerza cierto valor

válido (si el monto era < 0 lo consideramos como 0), o podemos implementar programación defensiva y chequear si la entrada está restringida y en ese caso devolver un error.

Similar al caso anterior, otra forma de subespecificar es no axiomatizar para ciertas entradas pero sin poner una restricción. Esto es un problema porque no sabemos cuál es el resultado para esas operaciones, y por lo tanto sería imposible diseñar los algoritmos del TAD.

Hay otra forma de subespecificar que es más sutil pero muy útil. Sirve para posponer la definición de algunos aspectos funcionales de las operaciones hasta la etapa de diseño e implementación. Cuando estamos escribiendo un TAD, hay partes del problema que quizás no tenemos definición aún, pero que tampoco son indispensables para modelar el comportamiento.

Por ejemplo, supongamos que estamos modelando un examen final de AED2 el cual se toma de forma oral, y tenemos una operación para llamar al siguiente alumno al aula. Durante el modelado no sabemos aún qué criterio se va a utilizar para llamar a los alumnos. Podría ser alfabético, por DNI, por LU, etc. Si en la axiomatización de llamar al siguiente alumno hacemos algo así: `prim(ordenarAlfabeticamente(alumnosQueRinden))` estamos prescribiendo exactamente la forma en que se llaman a los alumnos. Pero aún no sabemos cuál es el criterio! Podemos entonces intencionalmente subespecificar el problema y decir `dameUno(alumnosQueRinden)` para luego definir el criterio exacto durante la implementación.

Ejercicio 2

Explicar qué algoritmos de sorting pueden pararse en medio de un ordenamiento y tienen resultados parciales ordenados.

SelectionSort

Después de procesar x elementos, efectivamente tenemos los primeros x elementos ya en su posición final en el arreglo ordenado. Si miramos el invariante de este algoritmo se puede ver fácilmente que vale la propiedad, pues el invariante justamente nos dice que en la iteración i -ésima, el arreglo está ordenado desde su inicio hasta la posición i -ésima.

InsertionSort

Este caso es similar al SelectionSort, excepto que si frenamos después de procesar x elementos, solo podemos garantizar que esos elementos están ordenados relativamente entre sí. No necesariamente corresponden a los primeros x elementos del arreglo ordenado, ya que en futuras iteraciones podemos encontrar algún elemento el cual es swapeado hasta el inicio del arreglo si resulta que es menor que todos los x elementos ya ordenados.

HeapSort

Si se frena el algoritmo durante la etapa inicial de armado del heap a partir del arreglo de entrada, no podemos garantizar nada. Una vez armado el heap, podemos dar ciertas garantías dependiendo del tipo de HeapSort. Suponiendo que frenamos después de procesar x elementos:

- HeapSort “in place” (memoria constante), orden ascendente: Los x elementos más **grandes** van a estar en sus posiciones correctas en el final del arreglo.
- HeapSort “in place” (memoria constante), orden descendente: Los x elementos más **chicos** van a estar en sus posiciones correctas en el final del arreglo.
- HeapSort con memoria adicional, orden ascendente: Los x elementos más **chicos** van a estar en sus posiciones correctas en el comienzo del arreglo secundario donde se arma el resultado.
- HeapSort con memoria adicional, orden descendente: Los x elementos más **grandes** van a estar en sus posiciones correctas en el comienzo del arreglo secundario donde se arma el resultado.

MergeSort

No podemos obtener un resultado parcial si frenamos en el medio.

QuickSort

No podemos obtener un resultado parcial si frenamos en el medio.

CountingSort

Si frenamos durante la etapa de counting, no podemos garantizar nada. Pero si frenamos durante la etapa de armado del resultado final, después de procesar x elementos efectivamente vamos a tener los primeros x elementos ordenados.

Ejercicio 3

Implementar el algoritmo Floyd usando la técnica Divide & Conquer. Dado un árbol completo T , se debe retornar un árbol H con los mismos elementos que T , pero que cumpla el invariante heap. Se puede asumir que ya tiene implementados **SiftDown** y **SiftUp** con la complejidad adecuada. Dar la complejidad y justificar.

Planteamos el algoritmo para un max heap. El caso para min heap es análogo.

BuildMaxHeap(in T : árbol) \rightarrow out H : árbol

```

1: if  $T.size \leq 1$  then
2:    $H \leftarrow T$ 
3: else
4:    $H.root \leftarrow T.root$ 
5:    $H.left \leftarrow \text{BuildMaxHeap}(T.left)$ 
6:    $H.right \leftarrow \text{BuildMaxHeap}(T.right)$ 
7:    $\text{SiftDown}(H)$ 
8: end if
```

▷ Un árbol de un solo nodo (o ninguno) ya es un heap

▷ $O(\log(n))$

Complejidad

Utilizamos el teorema maestro para calcular la complejidad.

$$T(n) = 2T(n/2) + f(n)$$

Sea $a = 2$, $b = 2$, $f(n) = \text{SiftDown}(n) = O(\log(n))$.

Veamos si $f(n) = O(n^{\log_b(a)-\epsilon})$ para algún $\epsilon > 0$.

Tomando $\epsilon = 0,5 \Rightarrow O(n^{\log_b(a)-\epsilon}) = O(n^{\log_2(2)-0,5}) = O(n^{0,5}) = O(\sqrt{n})$.

Como $f(n) = O(\log(n)) \subset O(\sqrt{n}) \Rightarrow T(n) = \Theta(n^{\log_b(a)}) = \Theta(n)$ por el caso 1 del teorema maestro.

La complejidad de BuildMaxHeap resulta $\Theta(n)$.

Ejercicio 4

¿Qué pasa si en hashing doble h_1 es una constante o si h_2 es una constante?

Hashing doble es una forma de direccionamiento abierto, en donde todas las claves van a parar adentro del HashTable T , obteniendo la posición de la siguiente forma:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod |T|$$

h_1 constante

$$h(k, i) = (c + ih_2(k)) \bmod |T|$$

El primer hash de cualquier clave siempre resulta c pues $i = 0$ inicialmente. Por lo tanto, una vez que hayamos insertado la primer clave, todas las futuras claves van a colisionar en el primer intento y tendremos que realizar un barrido para encontrar su posición. Dado una clave k que colisionó, $h_2(k)$ resulta constante para esa clave en particular (las funciones de hash son determinísticas), y por lo tanto el barrido realiza saltos de $h_2(k)$ posiciones.

Puede haber aglomeración secundaria si h_2 produce el mismo valor para dos claves distintas, ya que en ese caso van a tener la misma secuencia de barrido a partir de la colisión inicial con $i = 1$. Pero si h_2 produce valores distintos para 2 claves, ante una colisión ya no necesariamente va a ser necesario realizar la misma secuencia de barrido hasta encontrar una posición libre pues los saltos de los barridos de las 2 claves serían distintos (pero siguen siendo barridos lineales y existe la posibilidad de generar secuencias de barrido iguales dependiendo de la constante usada y el tamaño $|T|$).

h_2 constante

$$h(k, i) = (h_1(k) + ic) \bmod |T|$$

La función de hash genera algo muy similar al barrido lineal en donde la posición inicial está determinada por la clave, y el barrido realiza saltos determinados por la constante $c = h_2$. A diferencia del caso anterior, acá la clave solo determina la posición inicial y el salto durante el barrido está fijo para todas las claves por igual.

Puede haber aglomeración primaria, ya que cualquier clave que colisiona con la aglomeración, va a seguir colisionando durante el barrido hasta llegar al “final” de la misma.

Conclusión

Si utilizamos una constante para h_1 o h_2 en una función de hashing doble, en esencia lo que sucede es que degradamos la función a un hashing simple con barridos lineales. En ambos casos sucede que para la mayoría de las constantes no vamos a poder asegurar que la función de hash produzca una permutación de todas las posiciones posibles de T .